Lasagne: A Static Binary Translator for Weak Memory Model Architectures

Rodrigo C. O. Rocha
University of Edinburgh
United Kingdom

Dennis Sprokholt*
TU Delft
Netherlands

Martin Fink
TU Munich
Germany

Redha Gouicem
TU Munich
Germany

Tom Spink
University of St Andrews
United Kingdom

Soham Chakraborty
TU Delft
Netherlands

Pramod Bhatotia
TU Munich
Germany

Abstract
The emergence of new architectures create a recurring challenge to ensure that existing programs still work on them. Manually porting legacy code is often impractical. Static binary translation (SBT) is a process where a program’s binary is automatically translated from one architecture to another, while preserving their original semantics. However, these SBT tools have limited support to various advanced architectural features. Importantly, they are currently unable to translate concurrent binaries. The main challenge arises from the mismatches of the memory consistency model specified by the different architectures, especially when porting existing binaries to a weak memory model architecture.

In this paper, we propose Lasagne, an end-to-end static binary translator with precise translation rules between x86 and Arm concurrency semantics. First, we propose a concurrency model for Lasagne’s intermediate representation (IR) and formally proved mappings between the IR and the two architectures. The memory ordering is preserved by introducing fences in the translated code. Finally, we propose optimizations focused on raising the level of abstraction of memory address calculations and reducing the number of fences. Our evaluation shows that Lasagne reduces the number of fences by up to about 65%, with an average reduction of 45.5%, significantly reducing their runtime overhead.

*The first two authors contributed equally to this work.

CCS Concepts: • Software and its engineering → Compilers; • Theory of computation → Formal languages and automata theory.

Keywords: Binary Translation, Memory Model, Compiler

ACM Reference Format:

1 Introduction
The landscape of hardware is shifting with the evolution of new computer architectures. Recently, processors based on the Arm and RISC-V instruction set architectures (ISAs) are emerging in consumer devices to servers running in data centers, disrupting the dominance that x86 once had [66?]. There are many benefits to using these modern architectures, including increased performance, better power efficiency [24], and better license stability.

However, a major challenge is ensuring that existing legacy applications can still continue to work on these new architectures. They provide different architectural features and memory consistency models. Porting an application can be as easy as recompiling for the new architecture, but older applications may not have the source-code available, or they may contain hard-coded architectural intrinsics that make a source-code based recompilation approach non-viable [?].

Static Binary Translation (SBT) is a process for automatically rewriting, ahead-of-time, the machine code from the original architecture to a target architecture. Because SBT works on the machine code itself, access to the original source-code is not required. Crucially, the translation must preserve the semantics of the original binary, as specified by the original architecture, whilst also optimizing the target binary in
a discernible way. Although SBT tools have gained popularity [25, 26, 75, 76], their support of several advanced architectural features is often limited. For example, Microsoft’s binary lifting prototype, called mctoll, was unable to lift the programs used in our evaluation.

Furthermore, these SBT tools are also unable to translate concurrent binaries [9, 13, 74]. This is due to the mismatches of the weak memory consistency model in different architectures, which governs the valid orderings of memory accesses. To address this problem, the translation tools must reason about the consistency models for correct and efficient translation.

In this paper, we address the challenge of developing efficient translation between x86 and Arm concurrency semantics through LLVM’s intermediate representation (IR). The x86 and Arm architectures have different memory ordering semantics, which results in different memory ordering rules. Therefore, we need a concurrency model for the LLVM primitives that enables precise mapping schemes between LLVM and these architectures, while also allowing for code transformations. We note that none of the existing concurrency models [18, 19, 30, 33, 52] suffice to satisfy all these requirements. Hence, to bridge this gap, we propose LIMM (LLVM IR Concurrency Memory Model). We use this model to design precise mapping schemes and prove their correctness.

We implement our approach in Lasagne, an end-to-end static binary translator. We extensively evaluate the effectiveness of Lasagne using the Phoenix multi-threaded benchmark suite [57]. Our evaluation shows that Lasagne reduces the number of fences by up to 65%, with an average reduction of 45.5%; thus, significantly minimizing the fence overheads. Overall, our paper makes the following key contributions:

- **Static binary lifting** (§ 4): First, we build a Binary Lifting tool capable of lifting concurrent binaries to the LLVM IR, while supporting several challenging architectural features. Our contributions have been merged into Microsoft’s mctoll2, as these functionalities are not fully supported by existing binary lifting tools.

- **IR refinement** (§ 5): Secondly, once the binary is lifted to the LLVM IR, we propose IR refinements to enable subsequent optimizations. Our IR refinement strategies are based on peephole optimizations and pointer parameter promotion. The IR refinement not only enables the standard LLVM optimizations for the target architecture, but it also aids in a significant reduction in the number of fences.

- **LLVM IR concurrency memory model** (§ 6–8): Lastly, we propose the LLVM IR’s concurrency model, named LIMM (§ 6). Based on LIMM, we design formally verified precise mapping schemes and also prove the correctness of the safe transformations in Agda (§ 7). We implement these mappings and optimizations in Lasagne (§ 8).

---

1Artifact: https://doi.org/10.5281/zenodo.6408463
2Microsoft’s mctoll: https://github.com/microsoft/llvm-mctoll

---

![Figure 1](image-url) Non-SC outcome $a = b = 0$ of SB program is allowed in x86 and Arm. Outcome $a = 1, b = 0$ of MP program is disallowed in x86 but allowed in Arm.

## 2 Background and Motivation

### 2.1 Concurrency in Architectures

A prevalent programming paradigm in modern multi-core architectures is *shared memory concurrency*, where concurrently running threads communicate through shared memory accesses. These architectures provide the following concurrency primitives: (1) load (ld) that reads from memory, (2) store (st) that writes to memory, (3) atomic read-modify-write (RMW), and (4) fence operations to order memory accesses.

**Weak memory concurrency in architectures.** Concurrent programs are usually understood by execution interleaving, where shared memory accesses in each thread execute in program order, and threads interleave arbitrarily. This execution model results in sequential consistency (SC) [34]. However, many architectures exhibit additional program behaviors which cannot be explained by interleaving alone, mainly due to out-of-order execution. These additional non-SC behaviors are known as *weak memory* behaviors.

**SB** shows an example comparing SC and weak memory. While both architectures exhibit certain common weak memory behaviors, their weak consistency models vary significantly. For instance, in x86, shared read-read and write-write memory access pairs are ordered, which is not the case in Arm. As a result, a program may exhibit different behaviors on different architectures. For example, the outcome $a = 1, b = 0$ of the MP program in Figure 1 is disallowed in x86 but allowed in Arm. These subtle differences affect the correct translations between architectures, and hence the architectures’ concurrency models require careful analysis.

**x86.** x86 provides mov instructions to perform ‘load from’ (ld) and ‘store to’ (st) operations. x86 also has a number of operations for RMW accesses. For instance, the cmpxchg instruction performs atomic compare-and-exchange operation on a memory location. Finally, x86 provides the MFENCE instruction that prevents memory accesses from being reordered across it.

**Arm.** Arm provides regular load (ld) and store (st) accesses, and load-linked (ll) and store-conditional (sc) pairs to construct RMW i.e. $\text{RMW} \triangleq \text{ll} ; \text{cmp} ; \text{bc} ; \text{sc} ; \text{lb} ; \text{fence}$: where cmp and bc are compare and jump instructions [33, 52]. Arm provides multiple fences such as DMBFF (full fence), DMBLD that prevent load-load and load-store pairs from being reordered, and DMBST that only orders store-store pairs. Arm also has release and acquire accesses, which act as half fences. We handle these accesses in details in Appendix A.
well as integer-based ones. The main difficulty in supporting mctoll lifts the x86 program in Figure 2a to the LLVM current programs may lead to erroneous program behavior. Hence, using them to translate concurrent programs may not always be correct for concurrent programs [17, 48, 63, 72]. We note the state-of-the-art SBT tools are written for sequential control flow graph. Several state-of-the-art binary lifters target an intermediate representation, e.g. LLVM IR, to ease this process [9, 13, 15, 20, 27, 68, 69, 74, 76]. Binary translators can operate statically or dynamically. The former allows more aggressive optimizations at the cost of being unable to handle dynamic jumps known at run time only. The latter handles those jumps but usually lifts at the basic block granularity, hindering optimization capabilities. Lifting full programs is a challenging and laborious task. Most existing tools are incapable of lifting the programs we use in our evaluation because they lack support for floating-point operations, SSE-based packed instructions as well as integer-based ones. The main difficulty in supporting these operations stems from the lack of abstract information in the source binary regarding types, control flow or function calls. Additionally, these lifting tools primarily target sequential programs and do not handle concurrency, i.e., they ignore the differences in memory consistency models altogether.

Limitations. Lifting full programs is a challenging and laborious task. Most existing tools are incapable of lifting the programs we use in our evaluation because they lack support for floating-point operations, SSE-based packed instructions as well as integer-based ones. The main difficulty in supporting these operations stems from the lack of abstract information in the source binary regarding types, control flow or function calls. Additionally, these lifting tools primarily target sequential programs and do not handle concurrency, i.e., they ignore the differences in memory consistency models altogether.

2.2 Binary Lifting

Our approach is based on static binary translation. Similar to modern compilers, the architecture of modern static binary translators also have a 3-phases structure. In both cases, their first phase (the compiler frontend and binary lifter) translates the input program to an intermediate representation (IR), e.g., LLVM IR. This IR code is then optimized and finally compiled down to its final binary format for a given architecture. There are two key benefits to this approach: First, the lifted code can be re-targeted to multiple architectures. Second, existing optimizations directly used on the lifted code.

The two fundamental principles of compilation [71] also apply to binary translation. First, binary translation must preserve the semantics of the input program, and second, the output program must be optimized in a discernible way. In this paper, we address both principles for concurrent programs.

State-of-the-art binary lifters. Lifting the source binary requires the binary translator to correctly map target to source instructions, discover global values, and reconstruct the control flow graph. Several state-of-the-art binary lifters target an intermediate representation, e.g. LLVM IR, to ease this process [9, 13, 15, 20, 27, 68, 69, 74, 76]. Binary translators can operate statically or dynamically. The former allows more aggressive optimizations at the cost of being unable to handle dynamic jumps known at run time only. The latter handles those jumps but usually lifts at the basic block granularity, hindering optimization capabilities.

Limitations. Lifting full programs is a challenging and laborious task. Most existing tools are incapable of lifting the programs we use in our evaluation because they lack support for floating-point operations, SSE-based packed instructions as well as integer-based ones. The main difficulty in supporting these operations stems from the lack of abstract information in the source binary regarding types, control flow or function calls. Additionally, these lifting tools primarily target sequential programs and do not handle concurrency, i.e., they ignore the differences in memory consistency models altogether.

2.3 Motivation: Translation for Concurrent Binaries

It is well-known that any transformation (mapping or optimization) written for sequential programs may not always be correct for concurrent programs [17, 48, 63, 72]. We note that the state-of-the-art SBT tools are written for sequential programs [9, 13, 15, 68, 74]. Hence, using them to translate concurrent programs may lead to erroneous program behavior.

As a concrete example, consider the translation in Figure 2. mctoll [74] lifts the x86 program in Figure 2a to the LLVM IR in Figure 2b, where it translates the shared variable accesses in x86 to non-atomic accesses. Next, LLVM reorders the shared memory non-atomic accesses (NA) and generates the optimized IR in Figure 2c. Finally, LLVM generates the Arm program in Figure 2d that may exhibit program outcomes that were originally not allowed in x86.

The error results from the lack of reasoning about concurrency at the IR level. To do so, the IR needs a concurrency model. Thus, the combination of mctoll and LLVM raises a question: What is the concurrency model of the IR?

A naive answer would be to insert full fences before or after all memory accesses to preserve correctness. However, full fences are costly in terms of performance and restrict a number of optimizations. To perform correct and efficient translation of concurrent programs, we require a concurrency model for the IR which fulfills the following desired properties:

- **Precise mapping schemes.** The IR concurrency model must facilitate precise mapping schemes from source to the IR as well as from IR to the target.
- **Optimized.** The IR should allow common transformations including shared memory access reordering, elimination and redundant fence elimination. The correctness of these transformations ensures that a compiler can safely apply the respective compiler optimizations on the IR.

3 Overview

Our approach targets statically translating an existing binary from a strong to weak memory model architecture. Figure 3 shows the overall workflow of LASAGNE, our end-to-end static binary translator between x86 and Arm. The x86 architecture employs a strong Total Store Ordering (TSO) model whilst Arm implements a weaker memory model [4, 5, 55].

The key aspect in supporting strong-to-weak binary translation is the strategic placement of memory fences to correctly emulate the ordering behavior of the source architecture. Our overarching goal is to support correct and optimized placement of fences, so that we emulate the source architecture faithfully, without introducing run-time overheads.

#1: Binary lifting. First we lift x86 binaries into LLVM bitcode. This is achieved by progressively raising the level of abstraction of the machine code, through multiple passes over the code and with the help of different IRs. The main challenge in binary lifting comes from having to reconstruct, from the machine, higher-level abstractions that have been lost in the compilation process. While lifting the binary to LLVM IR, it is important to identify these abstractions in order to enable
more aggressive optimizations by the following stages. We address these challenges in Section 4.

#2: IR refinement. Once the binary is lifted to the LLVM IR, we perform some refinements to the code to better enable the upcoming optimizations. In this paper, we are focused on exposing pointer types throughout the lifted code, instead of handling integer-based raw memory addresses. Our approach is two-fold: First, we propose some peephole optimizations that reconstruct this information within functions; Second, we propose pointer parameter promotion, which replaces integer parameters that are used as raw memory addresses with an appropriate pointer type. We discuss the IR refinement process in Section 5.

#3: LIMM: IR concurrency model. As LASAGNE translates from a strong to weak memory model, it inserts memory fences to achieve correctness. Excessively inserting fences degrades performance, whereas too few (or too weak) fences will lead to incorrect behavior. Approaching the minimum number of fences requires a careful understanding of concurrency models in different architectures.

We introduce LIMM (LLVM IR Memory Model), which acts as LASAGNE’s formal concurrency model. LIMM extends the concurrency primitives in the LLVM IR. The semantics of LLVM non-atomic accesses differ from both the corresponding x86 and Arm load and store accesses. In x86, ld-ld, ld-st, st-st access pairs are ordered, whereas non-atomic load and store accesses are always unordered. The Arm concurrency model disallows the removal of false dependencies [55] as these dependencies enforce certain orders between memory accesses. In contrast, LLVM regularly removes false dependencies in various optimizations. To allow these optimizations, LIMM does not order any accesses based on dependencies. We describe the details of LIMM in Section 6.

#4: Translation correctness in LIMM. Based on LIMM, we define precise mapping schemes for translating between architectures, and reason about the correctness of the common transformations on LIMM. More specifically, we identify the safe/unsafe reordering of independent shared memory accesses and fences. We also identify safe elimination of redundant shared memory accesses. The main challenge is to formally prove the correctness of the mapping schemes, and the safe transformations. We discuss the mapping schemes and the transformations in Section 7.

#5: Implementing LIMM translations. We implement our mapping schemes in LASAGNE, which appropriately inserts fences into the refined LLVM IR. In particular, these schemes demand leading or trailing fences for shared accesses. In Section 7, we describe reordering and elimination transformations, which LLVM regularly performs; these remain correct in LIMM. We discuss the further details in Section 8.

4 Binary Lifting

In this section, we describe the binary lifting component, which we develop based on Microsoft’s mctoll [74]. Figure 4 shows the workflow for binary lifting, including the three different IRs used throughout this process, namely, MCInst, MachineInstr, and finally the LLVM IR. By lifting the binary to the LLVM IR, we are able to re-optimize the program, enabling us to exploit features that are specific of the target ISA [41, 47] or focus on a different objective function such as code-size reduction [21, 58–61].

First, the source binary is disassembled to an array of MCInst, which is the lowest-level IR in LLVM, working as an in-memory representation of the disassembled binary code. Afterwards, this monolithic array of MCInst is processed and control-flow graphs (CFG) are reconstructed using MachineInstr. These two low-level IRs are also used in the LLVM backend during code generation; however in the reverse order. Finally, the LLVM bitcode is generated, function by function, via multiple passes over the code, progressively lifting the level of abstraction, as shown in Figure 4. We detail each one of these components in the subsections below.

Our contributions to mctoll can be summarized as follows:

- We add support for floating-point arguments/return types and tail-calls.
- We add support for around 100 instructions (400 instruction variants), mainly SSE instructions, which were not supported before.
- We also add support for some of the affected flags, e.g., the parity flag.
- We implement additional x86 features such as global variables declared in header files, e.g., std. out.

In total, we submitted 71 pull requests, 63 of which have been merged into the mctoll repository. Although we added support for almost 400 x86 instruction variants (or around 100
unique instructions) and several other x86 features, mctoll is still a work in progress.

4.1 Function Type Discovery

Before lifting a function or its calls to a strongly-typed language, such as LLVM IR, we need to know its function type, i.e., the list of parameters and return types. This function type discovery is based on the calling convention, which dictates how parameters and return types are arranged in the registers and stack memory. In this paper, we focus on the System-V application binary interface (ABI) [?].

**Type discovery.** First, we describe how we derive the types of a value in a specific register. This type discovery approach is used for both the list of parameters and return type.

Values in general-purpose registers are of integer type, with the width of the register as the width of the raised type, e.g., EDI → i32. As a result, pointers are also raised as raw 64-bit integer values, since pointers – at the machine level – are always manipulated using the general-purpose registers. We discuss this in more detail in Section 5.

For SSE registers, we need a different approach as they may hold either floating-point (FP) or packed values [53, 54]. While we can derive the integer type from the general-purpose register alone, this is not possible for SSE registers. To derive the type of an SSE register, we need to analyze the instructions using it. We differentiate between two types of SSE instructions:

- **Packed instructions** operate on vectors of integers or FP values. The discovered type is a 128-bit wide vector, the type and count of elements depends on the instruction, e.g., ADDPD (add packed double) → <2 × double>
- **Scalar instructions** operate on single FP values. The discovered type is either a float or a double, e.g., ADDSD (add scalar double) → double

**Parameter discovery.** Detecting the parameters of a function involves a live variable analysis [6] of the registers’ usage in the CFG. If a live register is one of the conventional parameter registers specified by the System-V ABI, and has no reaching definition, it is considered to be a parameter register. The System-V ABI allows for up to six integer and eight SSE-based parameters to be passed via registers, while additional ones will be spilled onto the stack.

**Return type discovery.** Discovering the return type of a function involves determining if one of the conventional return registers, RAX or XMM0, is alive at all the exiting blocks of the CFG. The return type can then be derived from such a register, as explained in Section 4.1. For functions where no return register is found, the type is set to void. If a function has multiple exit paths, the largest sized return type will be used.

4.2 Instruction Translation

In the last two passes, all instructions are lifted to the LLVM IR. Moreover, while traversing the CFG, we need to keep track of the values stored in the registers and the processor status flags, so operands can be correctly translated. First, we only lift non-terminator instructions. Then, the final pass lifts the terminator instructions, connecting the CFG in the LLVM bitcode.

Each MachineInstr can be translated to zero, one, or more LLVM instructions. Some instructions, such as copy operations, only update the internal record of which value the destination register is currently holding, generating no instruction in the lifted code. Instructions such as addition, where a simple LLVM counterpart exists, translate to a single LLVM instruction. Lastly, instructions that implicitly set processor status flags will result in more than one LLVM instruction. If any of these lifted instructions are unnecessary, they become dead code, later eliminated by traditional LLVM optimizations.

Programs may also contain function calls, that do not translate like instructions, as they take an arbitrary number of arguments – possibly of different types – requiring more consideration during translation.

4.2.1 Translating function calls. In order to translate function calls, we examine the information gathered by the function type discovery and standard library header files. Similar to parameter discovery, the list of arguments passed in a function call also accounts for the conventional parameter registers specified by the System-V ABI. The LLVM values referenced by the parameter registers are passed as arguments to the function call. If the function returns a value, the tracking record of the return register is also updated accordingly.

**Call to variadic functions.** Variadic functions can have a different number of arguments passed in each call site. All parameter registers alive at the callsite are passed an argument.

For SSE registers, the System-V ABI requires that the number of SSE-based arguments must be set in the AL register while calling a variadic function. Most compilers, such as LLVM, often explicitly assign a constant to the AL register. We try to leverage this behavior whenever possible. If we find a reaching constant value for the AL register, we look specifically for that amount of SSE registers. Otherwise, we fall back to the same approach used for general purpose registers.

**Challenges with parameter ordering.** When general purpose and SSE registers are passed as arguments to a function, we cannot reconstruct precisely the order of arguments from the original source code. This can be problematic for some variadic functions that depend on a specific argument order, e.g., printf from the C standard library. We can only preserve the order of the arguments within each group of registers.

In this paper, we assume that arguments passed via general purpose registers come before those passed via SSE registers. While this is not an issue when recompiling to ARM64 or another architecture that has a similar distinction between the two register groups, this may be an issue in the general case.

4.2.2 Handling of SSE Register Values. As discussed in Section 4.1, SSE registers can be interpreted in several ways
depending on the instructions using them. Therefore, we need to cast the values stored in SSE registers accordingly.

For instructions operating only on the lower bits of an SSE register, zeroing the remaining bits, e.g., most scalar FP instructions, we ignore the zeroed upper bits. This is beneficial for consecutive SSE instructions that operate on the resulting value in a similar manner.

For instructions that also operate on higher bits, however, we cast the FP value to a vector type, zeroing the upper bits. Since casting happens implicitly in x86, without modifying any bit, we use bit-casts in LLVM. We have to handle three cases in particular, with src as the source type, dst as the destination type, and |x| as the bit width of a given type x.

1. \( |\text{src}| = |\text{dst}| \): In this case, a simple \texttt{bitcast} instruction is used to cast the value to the desired type.

2. \( |\text{src}| < |\text{dst}| \): Since we assume missing bits to be zero, we first create a 128-bit wide vector of the type \(<n \times \text{src}>, \) where \( n = \frac{|\text{dst}|}{|\text{src}|}. \) Then, we insert the source value into position 0 of the created vector and bitcast the value to the destination one.

3. \( |\text{src}| > |\text{dst}| \): Here, we first bitcast the source value to a bit vector of type \(<n \times \text{dst}>, \) where \( n = \frac{|\text{src}|}{|\text{dst}|} \) and return the element at position 0.

Although we currently only support SSE instructions, our solution could be generalized to AVX2 and AVX-512. However, in this paper, we focus on SSE instructions used in the context of scalar floating-point computations.

### 4.2.3 Handling the stack memory

\texttt{mctoll} reconstructs the stack memory of a lifted function by allocating a byte (i8) array. Offsets relative to the stack base pointer are translated to indexing operations on the lifted byte array. After raising all instructions in a function, the stack accesses are inserted with indices relative to the end of the stack.

## 5 IR Refinement

After lifting a program to LLVM IR, much information remains absent. For instance, in machine code, there is no difference between integer values and pointers, even less what type is pointed. However, this information can be represented in LLVM IR, and is invaluable to enable standard LLVM optimizations. Furthermore, this lack of information hinders our optimized fence placement algorithm. We aim to rediscover this information—in particular pointer arithmetic—after mapping to the IR, to enable optimizations.

Pointer parameters are lifted as 64-bit integer parameters and \texttt{inttoptr} instructions are used to convert integer-based addresses to pointer types just before a load or store instruction. Similarly, stack allocated memory has its base pointer converted to an integer via the \texttt{ptrtoint}.

In this section, we propose transformations that address this issue. Our approach is twofold. First, we use peephole optimizations to rewrite integer-based address computations to

\[
\text{Rule 1: Pointer casting}
\]

\[
\begin{align*}
\%0 &= \text{ptrtoint} \ i8^* \ %\text{stacktop} \ to \ i64 \\
\%\text{RBP} &= \text{inttoptr} \ i64 \ %0 \ to \ i32^* \\
\%\text{RBP} &= \text{bitcast} \ i8^* \ %\text{stacktop} \ to \ i32^* \\
\end{align*}
\]

\[
\Rightarrow
\begin{align*}
\%\text{RBP} &= \text{inttoptr} \ i64 \ %0 \ to \ i32^* \\
\%\text{RBP} &= \text{bitcast} \ i8^* \ %\text{stacktop} \ to \ i32^* \\
\end{align*}
\]

\[
\text{Rule 2: Stack offset}
\]

\[
\begin{align*}
\%\text{tos} &= \text{ptrtoint} \ i8^* \ %\text{stacktop} \ to \ i64 \\
\%\text{RBP} &= \text{add} \ i64 \ %\text{tos}, \ %0 \\
\%\text{RBP} &= \text{inttoptr} \ i64 \ %0 \ to \ i32^* \\
\%0 &= \text{getelementptr} \ i8, \ i8^* \ %\text{stacktop}, \ i64 \ %0 \\
\%\text{RBP} &= \text{bitcast} \ i8^* \ %0 \ to \ i32^* \\
\end{align*}
\]

\[
\Rightarrow
\begin{align*}
\%0 &= \text{add} \ i64 \ %\text{arg}, \ %8 \\
\%\text{RBP} &= \text{inttoptr} \ i64 \ %0 \ to \ i32^* \\
\%0 &= \text{getelementptr} \ i8, \ i8^* \ %0, \ %0, \ i64 \ %8 \\
\%\text{RBP} &= \text{bitcast} \ i8^* \ %1 \ to \ i32^* \\
\end{align*}
\]

\[
\text{Rule 3: Parameter offset}
\]

\[
\begin{align*}
\%0 &= \text{add} \ i64 \ %\text{arg}, \ %8 \\
\%\text{RBP} &= \text{inttoptr} \ i64 \ %0 \ to \ i32^* \\
\%0 &= \text{getelementptr} \ i8, \ i8^* \ %0, \ %0, \ i64 \ %8 \\
\%\text{RBP} &= \text{bitcast} \ i8^* \ %1 \ to \ i32^* \\
\end{align*}
\]

\[
\Rightarrow
\begin{align*}
\%0 &= \text{add} \ i64 \ %\text{arg}, \ %8 \\
\%\text{RBP} &= \text{inttoptr} \ i64 \ %0 \ to \ i32^* \\
\%0 &= \text{getelementptr} \ i8, \ i8^* \ %0, \ %0, \ i64 \ %8 \\
\%\text{RBP} &= \text{bitcast} \ i8^* \ %1 \ to \ i32^* \\
\end{align*}
\]

**Figure 5.** Peephole optimizations on pointer arithmetic instructions. %stacktop is a pointer to a stack allocated memory, %arg is an integer parameter, %RBP holds the resulting memory address. Note that %RBP could be a pointer of any type.

their pointer-based counterparts. Second, all integer parameters only used as input to \texttt{inttoptr} are modified to a pointer type, possibly also introducing pointer casts where needed.

### 5.1 Exposing Pointers via Peephole Optimizations

The peephole optimizations comprise a collection of code patterns replaceable with another semantically equivalent piece of code. Figure 5 shows three examples of such patterns.

The patterns of code covered by these peephole rules mainly comprise integer-based arithmetic operations on the raw memory addresses. In this scenario, adding a number to a raw memory address is equivalent to offsetting an i8 pointer by that amount through the \texttt{getelementptr} (GEP) instruction. The GEP instruction performs address calculation only, and does not access memory. The address calculation is based on the size of the base type, the base pointer, and the integer indices provided. This equivalence is illustrated by transformation \( \odot \) in Figure 5. In order to simplify our peephole rules, we always use i8 as the base type of the GEP instructions, and then cast the resulting pointer to the expected pointer type.

### 5.2 Promoting Pointer Parameters

Since all pointer parameters are lifted as integer parameters, we need a way to identify when these integer parameters are actually used to represent a raw memory address. As illustrated by rule \( \odot \) in Figure 5, applying the proposed peephole optimizations can expose such integer parameters.

Our \textit{pointer parameter promotion} works as follows: for each integer parameter of a given function, we collect all its uses. If all its users are \texttt{inttoptr} instructions, we mark it for a pointer type promotion, otherwise, we keep it as is. We choose the pointer type depending on all the destination pointer types of the \texttt{inttoptr} instructions. If all of them have the same destination pointer type, we promote it to that type and simply
delete all its inttoptr instructions. Otherwise, we promote it to an i8 pointer, replacing all the inttoptr instructions to a bitcast to the appropriate destination pointer type.

6 LIMM: Concurrency Memory Model

In this section, we describe our next contribution. LASAGNE translates concurrent programs from x86 to the IR to Arm. We intend to ensure that the translation remains correct based on the formal concurrency models. We first describe the general components of axiomatic concurrency models. We then compare the x86 and Arm axiomatic models. Finally, we propose LIMM—an axiomatic concurrency model for the LLVM-IR.

6.1 Axiomatic Model of Concurrency

In axiomatic semantics, a program is represented by a set of executions. An execution consists of a set of events and relations between these events. Our notations are similar to the cat language used for defining axiomatic models for concurrency [4].

Event. An event is generated from the execution of a shared memory access or fence. An event is represented by \((id, tid, lab)\) where \(id\) is a unique identifier, \(tid\) is the thread identifier, and \(lab\) is the label of the event. A label \(lab = (op, loc, val)\) is a tuple where \(op\) denotes the corresponding memory access or fence operation. For memory accesses, \(loc\) denotes the corresponding memory location, while \(val\) denotes the read or written value. In case of fences, \(loc = val = \bot\). We respectively denote the set of read, write, and fence events by \(R, W, \) and \(F\). Every memory location is initialized at the start of the execution, represented by a set of write events (where \(tid = 0\)). Unless otherwise mentioned, memory locations are initialized to zero.

Relation. Various binary relations connect the events in an execution. Given a binary relation \(S\), we write \(S^r, S^t, S^s, S^{-1}\) to denote its reflexive, transitive, reflexive-transitive closures, and inverse relations respectively. Relation \(S_{imm}\) denotes the immediate relation: \(S_{imm}(a, b) \equiv S(a, b) \wedge \not\exists c \in S(a, c) \lor S(c, b)\). We write \([A]\) to denote the identity relation on a set \(A\), i.e. \([A](x, y) \equiv x = y \land x \in A\). Given two relations \(S_1\) and \(S_2\), we denote their composition by \(S_1; S_2\). In the model, an execution has the following relations between events:

- Relation program-order (po) is a strict partial order that denotes the syntactic order between the events. It is a strict total order on same-thread events.
- Relation reads-from (rf) relates every read event with the read events that read from it. Every read event reads from exactly one write event.
- Relation coherence-order (co) is a strict total order over same-location writes.
- A successful \(RMW\) results in an \(RMW\) relation between a pair of read and write events on the same location which are also in immediate-po relation; i.e. \(RMW \subseteq ([R]: po_{imm}([W]))|_{loc}\) holds. A failed \(RMW\) results in only a single \(R\) event. From these, we derive a number of other relations:

x86 axiom

\(GH\) hb* is irreflexive where

\[ppo \equiv \{((W \times W) \cup (R \times W)) \cup (R \times R)\} \cap po\]

\[implid \equiv po \cup \{At \cup U\}.po\]

where \(At \equiv \text{dom}(RMW) \cup \text{codom}(RMW)\)

\[hb \equiv ppo \cup \text{implid} \cup \text{U}\]

Arm axiom

\((\text{external})\) ob is irreflexive where

\[ob \equiv (\text{obs} \cup \text{aob} \cup \text{dob} \cup \text{obob})^+\]

\[\text{obs} \equiv \text{rf} \cup \text{co} \cup \text{U}\]

\[\text{aob} \equiv \text{rmw} \cup \ldots\]

\[\text{dob} \equiv \text{addr} \cup \text{data} \cup \text{ctrl} \cup [W] \cup \ldots\]

\[\text{bob} \equiv \{F\}.po \cup \{R\}.po \cup \{F\}.po \cup \{W\}.po \cup \{F\}.po \cup \{W\} \cup \ldots\]

Figure 6. Distinguishing axioms in x86 and Arm. Both models satisfy sc-per-loc and atomicity axioms (Section 6.2). Full Arm model is in Appendix A.

- Relation from-read (fr) connects a pair of read and write events accessing the same memory location: \(fr \equiv rf^{-1}: co\). If a read \(r\) reads-from a write \(w\), while write \(w'\) on the same location is \(co\)-after \(w\), then \(r\) and \(w'\) are in fr relation.

- We categorize the relations in an execution as either internal or external. If a relation \(S\) is between po-related events, then it is an internal-S relation and otherwise an external-S relation. For instance, we categorize the rf, co, fr relations in internal and external relations as follows:

\[rfi \equiv rf \cap po\]

\[coi \equiv co \cap po\]

\[coli \equiv co \cap po\]

\[fr \equiv fr \cap po\]

\[fr \equiv fr \cap po\]

\[rfe \equiv rf \cap po\]

\[cre \equiv co \cap po\]

\[cre \equiv fr \cap po\]

Execution. An execution \(X = (E, po, rf, co, rmw)\) is a tuple where \(X.E\) is the set of events and \(X.po, X.rf, X.co, X.rmw\) are set of \(po, rf, co, rmw\) relations between the events in \(X.E\).

From programs to executions. A program consists of a set of initialization writes on all shared memory locations followed by a parallel composition of threads. In a program, the concurrency primitives generate the following events and relations during an execution:

- A store (st) that writes value \(v\) on a shared memory location \(x\) generates an event with label \(W(x, v)\).

- A load (ld) that reads value \(v\) from a shared memory location \(x\) generates an event with label \(R(x, v)\).

- A successful \(RMW\) on \(x\) that reads value \(v_r\) and writes value \(v_w\) generates events with labels \(R(x, v_r)\) and \(W(x, v_w)\) that are \(rmw\)-related. If the \(RMW\) reads value \(v'\) and fails, it generates a single read event with label \(R(x, v')\).

- A full fence, e.g., MFENCE in x86 or DMBFF in Arm, generates an event with label \(F\).

Consistency. Axiomatic memory models define several consistency constraints, which capture architectural properties. An execution satisfying these constraints is consistent.
We use these events, relations, and definitions in all concurrency models, including those for x86 and Arm.

6.2 Comparing x86 and Arm Concurrency Models

We compare the axiomatic models of x86 and Arm [4, 5, 55]. The similarities and differences between these models guide us in defining the IR concurrency model.

Common axioms. Both the x86 and Arm architectures provide coherence and atomicity axioms. These properties are captured by the axioms described below.

Coherence. In an execution, coherence enforces SC-per-location: the memory accesses per memory locations are totally ordered. We capture this property as follows:

\[ (polloc ∪ rf ∪ co ∪ fr)^* \] is irreflexive. (sc-per-loc)

Atomicity. Suppose \( r \) and \( w \) are the read and write events generated from a successful RMW. These events are in \( \text{rmw} \) relation. If there exists a write event \( w' \) between \( r \) and \( w \) on the same location, such that \( \text{fre}(r, w) \) and \( \text{co}(w', w) \) hold, then the execution violates atomicity. Both x86 and Arm restrict atomicity violation with the (atomicity) axiom:

\[ \text{rmw} \cap (\text{fre} \cup \text{co}) = \emptyset \] (atomicity)

Differentiating x86 and Arm. In Figure 6, we state the distinguishing axioms between x86 and Arm. Axiom (GHB) in x86 and axiom (external) in Arm enforce a global order on x86 and Arm executions, respectively.

x86. In x86, the read-read, read-write, write-write access pairs are ordered by the ppo relation. The access pairs are also ordered by the \( \text{implid} \) relation, an intermediate F event, or the \( \text{rmw} \) relation. Relation x86-happens-before (hb) is defined using ppo, implid, \( \text{rf} \), co, fr relations. Finally, axiom (GHB) enforces a global order on x86 executions.

Consider the x86 program in Figure 9 and its execution with the outcome \( a = 1, b = 0 \). The execution contains a ppo \( \text{Ufr} \) cycle which is disallowed according to the (GHB) axiom.

Arm. We follow the axiomatic model of Arm from Pulte et al. [55]. Arm defines atomic-ordered-by (aob), dependency-ordered-before (dob), barrier-ordered-by (bob), and observed-by (obs) relations to define the (external) axiom in Figure 6.

Relations aob, dob, bob order po-related events. Relation obs orders same-location events in different threads. Relation aob is based on \( \text{rmw} \). Relation dob is defined using data, address, and control dependencies. These dependencies are captured by data, addr, and ctrl relations. Relation data, addr, and ctrl order a read to a po-successor write, read or write, and all events respectively. Relation bob is based on fences and synchronizing memory accesses. Relation obs is thread-external and relates same-location concurrent events. We use these relations to define relation ob. Finally, (external) axiom enforces a global order using the ob relation.

Unlike x86, the Arm model allows the execution with \( a = 1, b = 0 \) outcomes in the \( \mathbb{P}_{src} \) program in Figure 9.

6.3 IR Concurrency Model

Primitives. We use LLVM primitives, in particular \( \text{RMW}_{sc} \) atomic accesses (i.e. \( \text{RMW} \) with \( \text{seq}_{-cst} \) memory order), and the non-atomic load (\( \text{ld}_{na} \)) and store (\( \text{st}_{na} \)) instructions.

To order the non-atomic, LIMM uses various fences, such as Fsc (fence with \( \text{seq}_{-cst} \) memory order) to enforce a full fence like MFENCE in x86 and DMFF in Arm. LIMM introduces Frm and Fww into LLVM IR, which are similar to the DMBLD and DMBST fences in Arm. An Frm orders a load with its successor memory accesses (\( M \) refers to any memory access). Any Fww pair is ordered by an intermediate Fww fence.

Events. Given a program, we generate the following events and relations in an execution:

- For the non-atomic load and store accesses, we generate \( \text{R}_{na}(x, v) \) and \( \text{W}_{na}(x, v) \) events respectively.
- A successful \( \text{RMW}_{sc}(x, v', v) \) generates a pair of \( \text{R}_{sc}(x, v) \) and \( \text{W}_{sc}(x, v) \) events which are \( \text{rmw} \)-related. If it reads \( v' \) and fails, it generates a single \( \text{R}_{sc}(x, v') \) event. Relation \( \text{rmw} \) acts as a full fence similar to that of x86.
- The Fsc, Frm, Fww fences generate fence events with labels \( \text{F}_{sc}, \text{F}_{rm}, \text{F}_{ww} \) respectively.

Finally \( R = \text{R}_{na} \cup \text{R}_{sc} \) and \( W = \text{W}_{na} \cup \text{W}_{sc} \) hold in LIMM.

Relations. We define order (ord) and global-happen-before (ghb) relations in Figure 7. The ord relation orders po-related events. A pair of po-related events \( (a, b) \) are in ord relation in the following scenarios.

- (ord1) There is an intermediate \( \text{F}_{km} \) event where \( a \) is a read and \( b \) is a memory access.
- (ord2) \( a \) and \( b \) are writes with an intermediate \( \text{F}_{ww} \) event.
- (ord3) \( a \) is a \( \text{F}_{sc} \) event or an event generated from \( \text{RMW}_{sc} \).
- (ord4) Event \( b \) is an \( \text{F}_{sc} \) or a write generated from a successful \( \text{RMW}_{sc} \) access.

Note that we do not define any ordering based on dependencies in the IR. This is because LLVM may eliminate false dependencies. Such eliminations could introduce disallowed behavior, which would render the translations incorrect.

We can define a ghb relation on events across threads. On an execution graph, ghb\((a, b)\) implies a path from \( a \) to \( b \) by the combination of ord and external relations \( \text{fre}, \text{co}, \text{fr} \).

Axioms. We now define the consistency axioms in Figure 7. The (sc-per-loc) and (atomicity) axioms are also present in the x86 and Arm memory models. The (GOrd) axiom ensures a global order between events.
We propose the mapping schemes of the concurrency primitives between x86 and Arm through the primitives in IR. For each mapping scheme, we prove its correctness, as described in Theorem 7.1 and show that the scheme is precise.

**Theorem 7.1 (Mapping Correctness).** Let \( M_x \rightarrow M_t \) be a mapping scheme which generates target program \( P_t \) from the source program \( P_x \). The scheme is correct if for each consistent target execution \( X_t \in [\mathcal{P}_t]_M \) there exists a consistent source execution \( X_s \in [\mathcal{P}_s]_M \) such that \( \text{Behav}(X_t) = \text{Behav}(X_s) \).

**Definition 7.2 (Precise Mapping Scheme).** A correct mapping scheme is precise if for each fence used in the mapping, there exists a program where the fence is necessary and sufficient to preserve correctness, i.e., no weaker fence is sufficient and no stronger fence is necessary.

**x86 to Arm Mappings.** We obtain the x86 to Arm mappings in two steps: (1) x86 to IR and (2) IR to Arm in Figures 8a and 8b.
PLDI ’22, June 13–17, 2022, San Diego, CA, USA


Disallowed outcome \(a = b = 0\).

Disallowed outcome \(X = Y = 2\).

We study the correctness of the reordering of an adjacent shared memory access or fence pair \(a\) and \(b\). In Figure 11a, we mark the safe (✓) and unsafe (✗) reorderings. The non-atomic accesses can be reordered freely as performed by LLVM. Non-atomic accesses cannot reorder with an RMWsc operation, as that requires reordering with both events in the rmw, which is disallowed by the reordering rules in Figure 11a. A store can reorder with a successor Frm and a load can reorder with a predecessor as well as a successor Fww. Any pair of fences can reorder safely.

We prove Theorem 7.5 to establish the correctness of the safe reorderings. We show that a reordering does not remove any ord relation from the target while defining the corresponding source execution.

Memory access eliminations. Figure 11b enlists the safe redundant access elimination transformations for read-after-read (RAR), read-after-write (RAW), and write-after-write (WAW) transformations. A RAR or RAW transformation eliminates the following read accesses and uses the value read or written by the first access. In the WAW transformation, the first write is redundant and can be eliminated safely. In these three LLVM transformations, the shared memory accesses are adjacent. In the next three transformations, the memory accesses are non-adjacent, with fences between the memory accesses.

Speculative load introduction. LIMM also supports speculative load introduction where a shared memory load access is hoisted outside the conditional. The read value is used only when the conditional holds. This transformation is regularly performed in LLVM optimizations e.g. ‘SimplifyCFG’.

The transformation is correct on LIMM. A target execution contains the event corresponding to the speculative load which is absent in the source execution. However, in this case, the speculative load value remains unused and does not affect the program behavior.

Fence merging. We can safely merge a fence with an adjacent same or stronger fence. It is also safe to strengthen an Frm or Fww fence to a full fence Fsc. So given a pair of adjacent Frm and Fww fences, we can strengthen and merge them to create one Fsc, that is, \(F_{Frm} \cdot F_{Fww} \rightarrow F_{sc} \cdot F_{sc} \rightarrow F_{sc}\).

Proof strategy. We prove the theorems for the correct transformations in the following steps. Given a \(M_t\)-consistent execution \(X_t\) of \(\mathcal{F}_{tg}\), we (1) define a source execution \(X_s\) from \(\mathcal{F}_{sc}\). Following the mapping scheme, the memory accesses in \(X_s\) have corresponding accesses in \(X_t\). Similarly, the mapping ensures the \(sf\), \(co\), \(rmw\) relations in \(X_t\) correspond one-to-one to those in \(X_s\). Then, (2) we relate the \(X_s\) and \(X_t\) relations that are used in \(M_t\) and \(M_s\) and show that \(X_t\) satisfies the axioms in \(M_s\). Finally, we (3) show the \(X_s.co\) and \(X_s.rm\) relations match and hence \(X_s\) and \(X_t\) have identical behaviors. In mapping schemes the source and target models differ and for the transformations on the IR both \(M_s\) and \(M_t\) are LIMM model.

We mechanize the proofs for the mapping schemes and transformations in about 12,000 lines of Agda [2].

7.3 Adopting LLVM Semantics

LLVM performs various optimizations based on undefined behaviors [31, 35]. However, compiled x86 programs have a defined behavior. Hence, performing these optimizations on a lifted program may result in unsound translation. Considering this issue, LASAGNE assumes that mctoll’s lifting is correct and produces LLVM programs that are free from undefined behavior. LASAGNE also ensures sound translation for racy programs as LIMM allows the optimizations.
8 Implementing LIMM Translations

Lasagne is implemented on top of Microsoft’s mctoll binary lifting tool and the LLVM compiler framework, both open-source projects. Most of our contributions have already been accepted and incorporated in the main mctoll repository.

Precise fence placement. We enforce x86 to IR mapping from Figure 8a on the lifted code. We perform the fence insertion in two steps.
1. For every load and store, we explore the use-def chain of their pointer operand. In this exploration, we ignore bitcast and getelementptr operations, looking for a potential stack allocation. If the access is performed on a stack address, then no fence is inserted. Otherwise, the access is conservatively treated as a shared memory access and fences are inserted following the mapping scheme from Figure 8a.
2. We merge pairs of fences in the same basic block if there is no instruction in between that may access the memory.

LLVM optimizations. After fence placement, we apply the LLVM optimizations. These optimizations are crucial to eliminate unnecessary code produced in the binary lifting process. We evaluate the most impactful optimizations in Section 9.4.

Code generation. We implement IR to Arm mapping scheme (Figure 8b) in the LLVM backend to generate Arm code.

9 Evaluation

9.1 Experimental Setup

Experimental testbed. All x86 binaries are compiled on a machine with a quad-core Intel Xeon CPU E5-2650, 64 GiB of RAM, running Ubuntu 18.04.3 LTS. All Arm binaries are compiled on a machine with a 16-core Arm Cortex-A72, 32 GiB of RAM, running Ubuntu 18.04.5 LTS.

Benchmarks. We base our evaluation on the multi-threaded Phoenix benchmark suite [57], detailed in Table 1. The Phoenix benchmark suite contains scalar floating-point computations implemented as SSE instructions. We omit two programs from the suite as mcctoll lifts them incorrectly, resulting in a segmentation fault in one of them and an infinite loop in the other. Our initial investigation suggests that this issue arises from the way mcctoll handles some stack accesses through the base pointer. Fixing this would have required to rewrite most of mcctoll’s stack handling code.

Methodology. To minimize the effect of measurement noise, we repeat all experiments 25 times. We report the average values and their 95% confidence intervals.

Table 1. Phoenix multi-threaded benchmark suite.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Abbrev.</th>
<th># Functions</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>histogram</td>
<td>HT</td>
<td>4</td>
<td>171</td>
</tr>
<tr>
<td>kmeans</td>
<td>KM</td>
<td>7</td>
<td>235</td>
</tr>
<tr>
<td>linear_regression</td>
<td>LR</td>
<td>2</td>
<td>120</td>
</tr>
<tr>
<td>matrix_multiply</td>
<td>MM</td>
<td>3</td>
<td>179</td>
</tr>
<tr>
<td>string_match</td>
<td>SM</td>
<td>5</td>
<td>205</td>
</tr>
</tbody>
</table>

9.2 Overall Runtime Performance

Figure 12 shows our main performance results comparing all five versions described in Section 9.1. We report the normalized run time with respect to the Native version.

As expected, the Native version is the fastest one. This version has the benefit of compiling directly from the source code to the final target architecture. Meanwhile, the other versions must be translated in a conservative manner, following the semantics of the x86 architecture, thus naturally introducing some run time overheads. The unoptimized Lifted version is the slowest among all evaluated versions since it includes a significant amount of unnecessarily lifted code representing processor flag computations and other x86-specific features. All this unnecessary overhead is optimized away in the Opt version. Finally, we have the two versions that include the novel optimizations proposed in this paper, namely, POpt and PPOpt. The version fully optimized by Lasagne, PPOpt, has the best run time among all other translated variants, with an average normalized run time of 1.51, compared to an average of 2.89 of the unoptimized Lifted version.

Lasagne offers a statistically significant improvement for the histogram (HT) and kmeans (KM) benchmarks. For KM, in particular, the Opt version has a run time of around 3.85 seconds while the fully optimized PPOpt version has a run time of around 2.95 seconds, i.e., a speedup of around 1.3×.
9.3 Understanding the Impact of Each Optimization

Peephole optimizations. Figure 13 shows the percentage of inttoptr and ptrtoint instructions that can be optimized away by our IR refinement strategies. Our results show that about half, 51.1% on average, of these integer-pointer casting operations are removed. Most of the remaining inttoptr instructions belong to two different scenarios: (i) an integer parameter that could not be promoted to a pointer type; (ii) some level of pointer indirection, e.g., a raw memory address loaded from memory or returned by a function call which is later converted to a pointer type.

Optimized fence placement. Figure 14 shows how the IR refinement translates to fewer fences in the lifted LLVM bitcode. Our fence merging rules alone are capable of reducing the total number of fences by an average of 6.3%. However, the IR refinement enables LASAGNE to reduce the total number of fences by an average of 45.5%. The IR refinement better exposes pointers used by load and store instructions, allowing our fence placement algorithm to avoid adding fences to operations involving the stack memory.

Figure 15 isolates the overhead reduction achieved by simply removing unnecessary fences in the unoptimized Lifted version. In this evaluation, we are excluding the impact of reducing the number of fences on other LLVM optimizations. The goal is to analyze the performance improvement of reducing the number of fences alone. Our results show that our fence merging rules have a significant impact on the histogram benchmark program while the IR refinement has a bigger impact on the kmeans benchmark program.

9.4 Overall Impact on Code Size

We analyze the difference in code size of the lifted LLVM bitcode with respect to the native version in Figure 16. All versions of lifted LLVM bitcode are larger than their native counterpart. The existing LLVM optimizations alone are capable of greatly reducing the size of the lifted code, with an 84.4% increase on average. However, the fully optimized version PPOpt is the smallest among all the translated versions, with a 68.2% increase on average. This reduction is a result of the combined effort of removing integer-pointer casting operations, inserting fewer fences, and the existing LLVM optimizations.

We also study the individual impact of optimization passes on the number of instructions on the kmeans program in Figure 17. We apply each optimization in isolation on the lifted LLVM bitcode after applying our IR refinement and the optimized fence placement. We report only the most impactful ones, as some of the optimizations either have no impact or result in a small code size bloat. However, some of the unreported optimizations might still be present in the
standard optimization pipeline, since they could improve run time performance. The four most impactful optimizations are: \texttt{instcombine}, LLVM’s general peephole optimizations; \texttt{dce} and \texttt{adce}, basic and aggressive dead-code elimination; and \texttt{liCM}, loop invariant code motion. These four optimizations alone reduce the total number of instructions, in the lifted LLVM bitcode, by at least 35%.

10 Related Work

Concurrency semantics. Correct and optimized compilation of concurrency in programming languages is a well studied problem [5, 11, 12, 19, 30, 33, 51, 52, 62, 65]. These models differ from LIMM in concurrency primitives and semantics. Considering the primitives, there is a semantic gap between the non-SC fences in C/C++/LLVM and Arm. Podkopaev et al. [52] propose an intermediate semantic model IMM that models C/C++ primitives. IMM use dependency-based ordering, which disallows many dependency-breaking LLVM optimizations. As our proposed model (LIMM) has no dependency-based ordering, we can safely apply those optimizations. What limits IMM for our use-case is two-fold: (1) Implementing the IMM-specified dependency-based-ordering checking is non-trivial. (2) LLVM optimizations have to be restricted to preserve correctness in the presence of dependencies.

In C/C++ models [12, 18, 19, 30, 33], data races result in undefined behavior. In LLVM, read/write race has defined behavior where the racy read returns undefined and write-write races result in undefined behavior [18]. LIMM is closer to hardware memory models and has no undefined behavior. Kang et al. [30] is an operational model and [19] uses event structure to reason about multiple executions together whereas LIMM follows per-execution based semantics.

Program transformations under weak memory models have also been vastly explored [17–19, 28, 30, 37, 48, 63, 64, 72]. We analyze transformation correctness on IR concurrency model and use the same model to enable mappings between two different memory models, namely, x86 and Arm. We show the differences between x86 and Arm memory models, and then propose and prove correct an precise mapping scheme between them through proposed IR model.

Memory semantics enforcement. Robustness or stability based approaches check, by exploring executions, if a given program is SC-robust/stable against weaker models, inserting fences otherwise [1, 3, 14, 16, 32, 36, 38–40, 45, 50, 67]. Instead, we define a mapping scheme for translation tools that is valid for all programs and enforces x86 model instead of SC.

Binary lifting. A number of static binary lifters target the LLVM IR [9, 13, 15, 68, 74] for analysis and transformation. Previous work provided correct translations for SIMD [25, 26] or floating point instructions [26, 76]. However, these translation tools do not support concurrency.

McSema is a static binary translator that uses LLVM as its intermediate representation. Although McSema significantly covers x86 features, it is unable to correctly handle concurrency. Our solution could also be implemented in other static binary translators such as McSema. However, we chose mctoll because it is fully open-source, whereas McSema requires third-party commercial components.

Dynamic binary translation. Binary lifting is also used in a dynamic context, with cross-architecture binary or system emulation [8, 20, 23, 26, 27, 44, 73, 75]. ArMOR [44] proposes a framework to specify, compare and translate between memory consistency models, in a over-conservative manner. However, it lacks support for dependence-based orderings and RMW accesses in Arm. Pico [20] follows the ArMOR translation for POWER to x86 translation. However, Pico does not provide any formal guarantee of correctness. Lasagne differs in its static approach, but we also analyze dependence-based orderings and RMW accesses, while also providing a precise mapping from x86 to ARMv8.

Rosetta 2 is a commercial tool for Apple Silicon [8] that translates x86 programs to the Arm ISA, where x86 ordering is enforced by Apple hardware [29]. Rosetta uses both static and dynamic translation. We are unable to compare with Rosetta 2, as it is not open-source and very little details have been made public. Concurrency-wise, Rosetta 2 handles the memory model mismatch by implementing both Arm and x86-TSO models in hardware, whereas Lasagne is a purely software-based solution. Moreover, Rosetta is platform-dependent (OS X and Apple M1 chips), preventing us from running our benchmarks targeting Linux ELF.

Peephole optimizations. Peephole optimization is a well-known technique used by optimizing compilers. Peephole optimizations identify certain code patterns and replace them with more efficient pieces of code [22, 46, 70]. Compilers may apply peephole transformations at different levels, for machine independent as well as for machine specific optimizations. Efforts have also been made to prove the correctness of peephole optimizations [42, 49].

11 Conclusion

In this paper, we present Lasagne, a static binary translator for weak memory model architectures. Lasagne is able to lift x86 binaries to LLVM IR and then compile it to Arm while enforcing the x86 memory ordering model. We provide formally verified mappings from x86 to LLVM IR to Arm and transformations on the IR, as well as peephole optimizations that drastically reduce the resulting binary’s size and enable LLVM optimizations. We evaluate Lasagne and show that it generates efficient code in terms of size and performance.

Acknowledgments

We thank S. Bharadwaj Yadavalli for reviewing our patches to mctoll. We thank James Bornholt for shepherding our paper. This work was supported by a UK RISE Grant.
References


A  x86 and Arm Concurrency Models

A.1 x86 Model [4, 5].

Given a shared memory location x,
- st that writes value v on x generates an event with label W(x,v).
- ld that reads value v from x generates an event with label R(x,v).
- RMW operations e.g. LOCK CMPXCHG or XCHG instructions perform atomic read-modify-write operations (RMW). A successful RMW on x that reads value o' and writes value v generates a pair of events e and e' with labels R(x,o') and W(x,v) respectively such that (e,e') ∈ [R];rmw;[W] holds. A failed RMW generates an R(x,o') event.
- An MFENCE generates an event with label F.

Relations. x86 defines following relations.
- Relation preserved-program-order (ppo).
  ppo ≜ ((W×W) ∪ (R×W) ∪ (R×R)) ∩ po
- Relation implid for fences.
  implid ≜ po ∩ (dom(rmw) ∪ F) ∪ [codom(rmw) ∪ F];po
- Relation happens-before (hb).
  hb ≜ ppo ∩ implid ∩ urs ∩ fr ∩ co

Consistency Constraints. x86 consistency constraints are as follows.
- (po ∪ fr ∪ fr ∪ co) is acyclic. (sc-per-loc)
- rmw ∩ (fre ∪ co) = ∅ (atomicity)
- hb is acyclic. (GHB)

A.2 ARMv8 Model [55]

- Load (ld) or load-linked (ll) that reads value v from x generates an event with label R(x,v).
- Store (st) or store-conditional (sc) that writes value v on x generates an event with label W(x,v).
- Acquire-read (ldA) or acquire-load-linked (llA) that reads value v from x generates an event with label A(x,v).
- AcquirePC-read (LDAPR) that reads value v from x generates an event with label Q(x,v).
- Release-write (stL) or release-store-conditional (scL) that writes value v on x generates an event with label L(x,v).
- Atomic read-modify-write operations are as follows.
  - A successful RMW generates [R];rmw;[W] and if an RMW fails then it generates an R event.
  - A successful RMWAL generates [A];rmw;[W] and if an RMWAL fails then it generates an A event.
  - Full fence DMBFF generates a F event.
  - Load fence DMBLD generates a F_L event.
  - Store fence DMBST generates a F_st event.
  - Control fence ISB generates an ISB event.

Relations. ARMv8 defines various relations.

Relations among events: A ⊆ R, Q ⊆ R, L ⊆ W. Dependency relations are same as ARMv7 except ISB is an event in ARMv8.
- Relation coherence-after (ca) orders read or write to a fr-after or co-after write.
  ca ≜ fr ∪ co
- Relation observed-by(oobs) constitute of thread external fre, fre, and coe relations.
  oobs ≜ fre ∪ coe ∪ fre
- Relation atomic-ordered-by(aob) is derived from rmw and rfi relations.
  aob ≜ rmw ∪ [codom(rmw)]; rfi; [A ∪ Q]

Relation dependency-ordered-before(dob) captures dependency based ordering among a pair of events in a thread
  dob ≜ addr ∪ data
  ∪ ctrl;[W]
  ∪ (ctrlU(addr);po;[L])
  ∪ addr;po;[W]
  ∪ (ctrlUdata);coe
  ∪ (addrUdata);rfi

Relation barrier-ordered-by(bob) orders events by fences or results from stronger memory accesses.
  bob ≜ po;[F];po
  ∪ [L];po;[A];
  ∪ [R];po;[F_St];po;
  ∪ [A ∪ Q];po
  ∪ [W];po;[F_st];po;[W]
  ∪ po;[L]
  ∪ po;[L];coe

Relation ordered-before (ob) is a transitive closure of obs, aob, dob, and bob relations.
  ob ≜ (obs ∪ dob ∪ aob ∪ bob) *

Consistency Constraints An well-formed ARMv8 execution X is consistent when:
- (X,po ∪ cnt ∪ C ∪ X,fr) is acyclic. (internal)
- X,ob is irreflexive. (external)
- X,rmw ∩ (X,fre;X,coe) = ∅ (atomicity)
B Arm to x86 Mapping

We get Arm to x86 mapping by combining Arm to IR and IR to x86 mappings discussed in fig. 18a and fig. 18b respectively.

B.1 Arm to IR

Following the mapping scheme in fig. 18a an Arm load access is mapped to a load along with a trailing Frm fence. An Arm store is mapped to a store in IR and an stL is mapped to an IR store with leading Fww and trailing Fsc fences. Any RMW in Arm is mapped to an RMW in IR. We map Arm full fence DMBFF to Fsc in IR. The weaker Arm fences generate no fence in the IR.

B.2 IR to x86

Following the mapping scheme in fig. 18b the load, store, RMW, and Fsc primitives in the IR are mapped to x86 load, store, RMW, and full fence accesses. The other fences in the IR generate no x86 access.

B.2.1 Arm to IR to x86. In fig. 18c we combine the mapping schemes of fig. 18a and fig. 18b. The resulting mappings are also precise.

The IR to x86 mapping introduce no fence for the memory access mappings. So the mapping scheme is precise. We show that the Arm to IR mapping scheme is also precise.

B.3 Examples

for precise fences in Arm to IR mapping scheme

We explain the leading and/or trailing fences introduced along the intermediate mapping:

\[ X = Y = 0; \]
\[ X = 1; \]
\[ a = Y; \]
\[ DMBFF; \]
\[ Y = 1; \]
\[ b = Y; \]
\[ Fsc; \]
\[ Fww; \]
\[ X = Y; \]
\[ a = Y; \]
\[ X = 1; \]
\[ a = Y; \]
\[ X = 1; \]
\[ a = Y; \]
\[ X = 1; \]
\[ a = Y; \]
\[ X = 1; \]
\[ a = Y; \]

The source Arm program disallows outcome \( a = 1, b = 0 \). The target IR program disallows the outcome. Without the intermediate Fww fence the outcome would be allowed. Hence the leading Fww fence is essential for stL mapping.

Fsc. Consider the following example for the Fsc fence in stL mapping:

\[ X = Y = 0; \]
\[ Fww; \]
\[ X = Y; \]
\[ Fww; \]
\[ X = Y; \]
\[ a = Y; \]
\[ X = 1; \]
\[ a = Y; \]
\[ X = 1; \]
\[ a = Y; \]

The source Arm program disallows outcome \( a = 0, b = 0 \). The target IR program disallows the outcome. Without the intermediate Fsc fences the outcome would be allowed. Hence the trailing Fsc fence is essential for stL mapping.

Frm. Consider the following example for the Frm fence in ld mapping:

\[ X = Y = 0; \]
\[ a = X; \]
\[ b = Y; \]
\[ Y = a + 0 + 1; \]
\[ X = b + 0 + 1; \]
\[ a = Y; \]
\[ X = 0; \]
\[ b = X; \]
\[ b = Y; \]
\[ X = b + 0 + 1; \]
\[ a = X; \]

The source program disallows \( a = b = 1 \) outcome in Arm. To disallow the same in the target program in the IR the Frm is required in Arm to IR mapping.

Note that Arm (version 8) does not remove false dependency [55]. In IR we eliminate false dependence as follows. In that case the fences preserve the required orders for correctness.
C Counter-examples of Unsafe Reorderings

In fig. 11a we display safe reorderings on LIMM. For the safe reorderings, we provide proofs.

We provide counter-examples for some of the unsafe reorderings.

**RM-fence.** An Frm cannot reorder with a preceding read instruction. Consider the following example:

\[
\begin{align*}
X &= Y = 0 \\
X_{NA} &= 1; \\
Y_{NA} &= 1; \\
\text{Frm; } b &= Y_{NA}; \\
\text{Frm; } a &= X_{NA}; \\
\end{align*}
\]

The target program allows the output \(a = b = 1\), while the source program disallows this.

**WW-fence.** An Fww cannot reorder with a preceding write instruction. Consider the following example:

\[
\begin{align*}
X &= Y = 0 \\
X_{NA} &= 1; \\
Y_{NA} &= 2; \\
\text{Fww; } b &= X_{NA}; \\
\text{DMBFF; } b &= X_{NA}; \\
\end{align*}
\]

The target program allows the output \(a = 2, b = 0\), which the source does not.

**RMW - RMW.** It is unsafe to reorder two RMW operations, even if they access different locations. Consider the following program.

\[
\begin{align*}
X &= Y = 0; \\
\text{RMW}_{sc}(X, 0, 1) &\rightarrow \text{Frm; } a = Y_{NA}; \\
\text{RMW}_{sc}(Y, 0, 1) &\rightarrow \text{Frm; } b = X_{NA}; \\
\end{align*}
\]

Outcome \(a = 1, b = 0\) is disallowed in the source program but allowed in the target program after the reordering of the \(\text{RMW}_{sc}\) accesses.